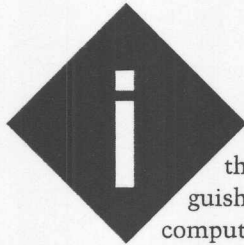


Who's in Control?

EMBEDDED TECHNIQUES

John Dybowski



It could be said that what distinguishes embedded computers from their

office-bound counterparts is the fact that the embedded variety are primarily designed for unsupervised operation. These things are forever finding their way into potentially troublesome situations where they are either dispersed about remote locations or placed into seemingly inaccessible predicaments. In any case, it seems wherever they end up the electrical and environmental conditions are usually less than favorable. In spite of these difficulties, we assume these devices will operate for extended periods without the need for human intervention.

Many different applications can be served, but the most demanding casts the embedded computer in the role of a system controller. Should such a control-oriented system malfunction, the results can range from mildly irritating to utterly destructive; there's nothing worse than a system controller running out of control. In the majority of cases, this loss of control can be attributed to electrically hostile conditions, and some external event can be blamed for the system malfunction. However, it pays to keep an open mind since the culprit may be indicative of some design inadequacy.

One of the worst offenders is the typical RC processor reset circuit. You will, no doubt, experience no trouble using such a circuit throughout the course of your product development. But chances are good that once you've moved your system away from the lab bench, you may begin to experience

new problems. Remember: it's an embedded controller's lot in life to be around equipment that wreaks havoc with the power grid. In such a setting, it's not uncommon to find out that your controller checks out every time some big motor kicks in. Apparently what's happening here is that the power dips to a point where loss of regulation occurs. Once V_{cc} drops out of spec, all bets are off. Unfortunately, a simple RC reset circuit that works so well on the lab bench is sufficiently primitive to be of no use at all in such a situation. Obviously, one way to solve this problem is to incorporate a decent reset circuit into the design...but it pays to look further.

Once you start examining all the real-world ills that can trouble your apparatus, it becomes obvious that the construction of a truly bullet-proof system can be extremely difficult. Fortunately, although most embedded computers operate under electrically hostile conditions, only a small percentage of these have to be truly impervious to any kind of interruption. Most systems are amenable to temporary disruptions as long as control can eventually be restored. Clearly, a single-chip implementation that only knows how to cold boot has quite different needs than a complex data collection system that is charged with collecting and retaining megabytes of data over extended periods of time.

Assuming the embedded computer is designed properly to begin with, some desirable attributes can be identified that can strengthen robust operation. These include a V_{cc} referenced reset circuit, a power-fail indicator/interrupt, and a watchdog to reset all subsystems if all else fails. Having mentioned that the DS2250 possesses the capability to operate under adverse conditions, it shouldn't be surprising to find that these features all come built-in.

DAMAGE CONTROL

The DS2250 generates an internal power-on reset without the need for external components. When V_{cc} is below the reset threshold, the DS2250 is held in an internal reset state. Once



In his continuing series on building

an embedded control system, John looks at reset circuits, power fail conditions, more battery management, and finally expands on his treatment of the LCD display interface.

new problems. Remember the embedded controller's around equipment that with the power grid. In it's not uncommon to your controller checks some big motor kicks what's happening here power dips to a point v regulation occurs. One of spec, all bets are off a simple RC reset circ well on the lab bench primitive to be of no u a situation. Obviously solve this problem is t decent reset circuit in design...but it pays to

Once you start ex real-world ills that ca apparatus, it becomes construction of a trul reset: system can be extrem Fortunately, although computers operate un hostile conditions, on percentage of these h impervious to any k power control register gives the user access to all kinds of features not found on a normal 8031 for dealing with reset and power fail conditions.

tion. Most systems a temporary disruption to the operating control can eventually reset generator Clearly, a single-chip reset before releasing that only knows how operation. This delay quite different needs counting 21,504 data collection systems, which with collecting and 11.95 ms when bytes of data over ex MHz. This time time.

Assuming the e a valid operational is designed properly it takes the crystal's some desirable attri operational motion. identified that can s the reset interval as operation. These in when operating from enced reset circuit, ecially indicator/interrupt, s powered reset all subsystems in trying Having mentioned power possesses the capabily under adverse condstem to be surprising to findakes little all come built-in. ms or

consider- The DS2250 genal" power-on reset witons. As external componen the reset below the reset thre market is held in an intern

erance

CONTROL REGISTER						
Register Address: 087H						
	D5	D4	D3	D2	D1	D0
R	PFW	WTR	EPFW	EWT	STOP	IDL

POR Previous reset initiated during power on sequence. Cleared to 0 when power on reset occurs. Remains at 0 until set to 1 by software. Can be read normally at any time. Can be written only by using Timed Access Register. PFW Indicates potential power failure in progress. Cleared to a 0 on power on reset. Can be read normally at any time. Not writable. WTR Set to 1 following Watchdog Timer timeout. Cleared to 0 after read of PCON reg. Set to 1 after Watchdog Timer Reset. Clear to 0 on power on reset. May be read normally at any time. Cannot be written.	PCON.3: EPFW Enable Pwr Fail Interrupt: Used to enable or disable Power Fail Interrupt. Initialization: Cleared to a 0 on any type of reset. Read Access: May be read normally at any time. Write Access: Can be written normally at any time. PCON.2: EWT Enable Watchdog Timer: Used to enable or disable Watchdog Timeout Reset. Initialization: Cleared to 0 on No-Vli power on reset. Unchanged during other types of reset. Read Access: May be read normally at anytime. Write Access: Can be written only by using Timed Access Register. PCON.1: STOP Stop: Used to invoke the Stop mode. Initialization: Cleared to 0 on any type of reset. Read Access: Can be read at anytime. Write Access: Can be written only by using Timed Access Register.
--	---

power control register gives the user access to all kinds of features not found on a normal 8031 for dealing with reset and power fail conditions.

on the reset duration, and in most cases will keep reset active for an excess of 1 second. In any event, these circuits work perfectly well in most fixed base applications, but just aren't appropriate for the type of system I'm developing here. Note that the DS2250's external reset pin is completely functional and can be used to place the DS2250 into reset at any time, but is unnecessary for implementing a normal power-up reset. Most designs will use the reset pin in conjunction with VSEN to force the DS2250 into bootstrap mode, which I'll be covering in a future column.

A built-in watchdog timer is provided as a means of restoring normal operation if the processor gets lost. The watchdog timeout period is equal to 122,880 machine cycles, which comes to about 133 ms at 11.0592 MHz. The nice thing about the DS2250's watchdog, unlike most outboard watchdog circuits, is that it can be turned on and off under firmware control. Personally, I wouldn't be without a watchdog, but depending on who's writing the code and what the code is doing, in some cases a watchdog could actually do more harm than good. In any event, it's useful to be able to keep the watchdog out of the way during initial firmware development.

All watchdog accesses must be performed using timed access, which is a method of protecting critical DS2250 bits and bytes from inadvertent modification.

INTERRUPT PRIORITY REGISTER							
Label: IP				Register Address: 0B8H			
D7	D6	D5	D4	D3	D2	D1	D0
RWT	—	—	PS	PT1	PX1	PT0	PX0
IP.7: RWT Reset Watchdog Timer: When set to 1, Watchdog Timer count reset. Writing 0 to bit has no effect. Initialization: Cleared to 0 on any reset. Read Access: Cannot be read. Write Access: Can be written only by using Timed Access Register.							

Figure 1b—An extra bit is added to the interrupt priority register to give special consideration to the DS2250's watchdog.

Certainly the system watchdog qualifies as a critical system resource. Timed access is accomplished by writing two bytes to the timed access register located at C7h. The first write must be the value AAh with the second being a 55h. After this sequence, the protected region may be accessed, but there's a limit on how long this access window remains open. When the initial AAh is written, two timers are initiated. The first timer allows two instruction cycles until the 55h is written. The second timer restricts access to the protected area to within four instruction cycles of the first write (the AAh). Note that since this timer is started before the writing of the 55H, the time remaining for the actual access depends on the type of instruction that was used to write the 55h. If a one-cycle instruction was used to write the 55h, then three cycles remain to modify the protected bits. If a two-cycle instruction was used to write the 55h, then there are only two cycles left.

The enable-watchdog-timer bit is located at PCON.2 and is referred to as EWT. Although this bit requires timed access to write, it can be read normally at any time. The write-only watchdog reset bit is at IP.7. Figure 1 shows how these new bit assignments are squeezed into the PCON and IP registers. Listing 1 illustrates how timed access is used to enable and reset the watchdog function.

It's often a good idea to have a power fail interrupt so you can take care of any loose ends during a power failure before the micro fades to black. Incorporating this capability can be a problem with the limited interrupt resources available on a standard 8031. You seldom have enough interrupts to begin with, and the two-level priority structure and global enable makes things even more difficult. Of course, even if you do have a loose interrupt available, you still have to construct a power monitoring comparator.

The DS2250 has the power monitoring circuitry built in and adds a new power fail warning (PFW) interrupt to the standard 8031 interrupt set that overcomes prior limitations. The PFW interrupt is enabled by

Listing 1—Writing to the watchdog must be done within time limits set by the time access register.

TA	EQU	0C7H	:TIMED ACCESS REGISTER
RWT	EQU	IP.7	:WATCHDOG RESET
EWT	EQU	00000100B	:ENABLE WATCHDOG TIMER
	MOV	TA, #0AAH	
	MOV	TA, #55H	:ENABLE TIMED ACCESS
	ORL	PCON, #EWT	:ENABLE WATCHDOG
	MOV	TA, #0AAH	
	MOV	TA, #55H	:ENABLE TIMED ACCESS
	SETB	RWT	:RESET WATCHDOG

setting the EPFW bit at PCON.3 to a 1. The corresponding power-fail warning flag is at PCON.5. Whenever V_{cc} drops below the low-voltage threshold, the PFW bit is set to a 1 and remains set until it is read by firmware. If the PFW interrupt is enabled, the processor will vector to location 2Bh on recognition of the low-voltage event.

To make this feature even more useful, the PFW interrupt is not controlled by the EA global interrupt enable bit. It can only be enabled or disabled by using EPFW, sort of a maskable nonmaskable interrupt. Furthermore, the PFW interrupt has the highest priority if it is enabled. The other interrupts can be programmed to either low or high priority using the standard two-level priority structure, but the PFW interrupt will interrupt any ISR if it is enabled.

One of the problems you pick up as a result of adding these recovery features is you must now make some conscious decisions on how to proceed if a problem occurs. Once you've taken steps to incorporating fault detection (and presumably tolerance) into your system, you've essentially relinquished the prerogative to be happy and dumb.

In the case of a reset event, the DS2250 lets you determine where you're coming from (kind of) so you can best approximate where you should be. This information is contained in some previously unused bits of the PCON special function register. Referring again to Figure 1, you can see that status bits exist which indicate that the most recent reset was a power-on reset or a watchdog timer reset. Additionally, if you are using the

power-fail interrupt capability, you can add your own indicators to signal a power failure incident.

Beyond just denoting the occurrence, the use of the power fail interrupt gives you capabilities that would not be available otherwise. Having an early warning of an impending power failure allows you not only to recover from an untimely loss of power after the event, but gives you a chance to orchestrate a organized, if not hurried, shutdown. But again, you must first decide what actually constitutes an orderly shutdown.

At one extreme you can enter into a benign code loop that tries not to do any harm while waiting for the power to either die or to return to normal. At the other extreme you might go so far as saving the processor's entire machine status in order to resume exactly where you left off on restoration of power. At first this sounds just like the thing, but in most cases the results prove to be less than desirable. In most real-time applications, there's a lot of setting up involved prior to actually processing any event. Then again, it's a pretty sure bet that the event of interest itself had passed into history long before operation was restored. Ironically, the type of processing that lends itself best to this type of resumption of operation is that which centers around performing computations and manipulations on stored data. This type of batch processing is not really the most common activity for an embedded computer.

As an alternative, you might want to record the system's operating state at the time of the power failure and then sort things out on power up. In

Listing 2—Miscellaneous LCD support services include the putchar function needed for driving the I²C-based 4 × 20 LCD panel.

```
#pragma LARGE CODE
#include "REG5000.h"

/* Defined bits for LCD control over the I2C bus */
#define Lcd0 0x42
#define Lcd1 0x44
#define LCDs 2
#define DEN 0x20
#define DRS 0x10

extern void InitLcd();
extern void ClearLcd();
extern void PositionCursor(unsigned char c);
extern void LCD_OUT(unsigned char c);

unsigned char Cursor[LCDs];
unsigned char ActiveLcd;

/* Putchar to LCD */
void putchar(unsigned char c)
{
    extern void DataWr(unsigned char c);
    unsigned char b;

    if (c == '\n') {
        b = Cursor[ActiveLcd];
        if (b < 20)
            PositionCursor(20);
        else if (b < 40)
            PositionCursor(40);
        else if (b < 60)
            PositionCursor(60);
        else
            PositionCursor(0);
    }
    DataWr(c);
    return;
}

/* Position LCD cursor */
void PositionCursor(unsigned char c)
{
    extern void CommandWr(unsigned char c);

    Cursor[ActiveLcd] = c;
    if (c > 60)
        CommandWr((c - 60) + 84 + 0x80);
    else if (c > 40)
        CommandWr((c - 40) + 20 + 0x80);
    else if (c > 20)
        CommandWr((c - 20) + 64 + 0x80);
    else
        CommandWr(c + 0x80);
    return;
}

/* Clear LCD and home cursor */
void ClearLcd(void)
{
    Cursor[ActiveLcd] = 0;
    CommandWr(1);
    return;
}

/* Write to LCD data register & handle cursor positioning */
static void DataWr(unsigned char c)
```

(continued)

this case, the operational state could be resumed or abandoned as appropriate. Naturally, there are as many approaches to fault recovery in embedded systems as there are embedded systems. Alas, this is one area that must often be reinvented to precisely suit the particular system.

LCD CONTROL

Last month I showed you some code for driving the ec.25's PC-based keyboard. Space limitations prevented me from finishing up with the display side of things, so now I'll wrap things up with a discussion of the LCD firmware support package. As with the keyboard driver, you will find all of the low-level functions required to handle a standard character-based LCD panel along with a way to hook into some standard C library functions. Since we picked up an ec.25-compatible getkey replacement last month, this time around we'll acquire a new putchar.

At the heart of just about every character-based LCD is the HD44780 LCD controller. And although it does have some idiosyncrasies, it is by any estimation an unbelievably successful design. To those who have employed small LCDs in their designs, the problems associated with driving these things are already understood. In any event, much has already been written on the subject, so I promise to be brief.

The fact that most LCDs operate under control of the HD44780 is, at the same time, both an advantage and a liability. It's an advantage, because once you've figured out how to use it, you're all set to handle just about any character-based LCD on the market. It's a liability since because it must be coerced to drive so many different display configurations, it has a multitude of options. Because of this, it is, out of necessity, somewhat quirky. Although there are a number of nagging problems associated with using the HD44780, perhaps the biggest turns out to be the lack of correspondence between physical and logical cursor positions. In actuality, this is not so much an LSI problem as an implementation problem, but nonetheless it is an affliction that is unique to LCDs.

I'll now describe some support functions that I've coded up using C. Although there isn't a one-to-one correspondence, I've already presented some assembler routines in a previous column that perform similar functions to those I'll show here. After all, how many different things can you do to an LCD, anyway? If you're interested in seeing how they compare against this month's offering, check out my column in issue 35, "Putting I²C Through Its Paces."

Something to keep in mind as you follow Listing 2 is that the code is written to support multiple LCD panels over the same two-wire bus. This capability only results in a slight increase in code complexity.

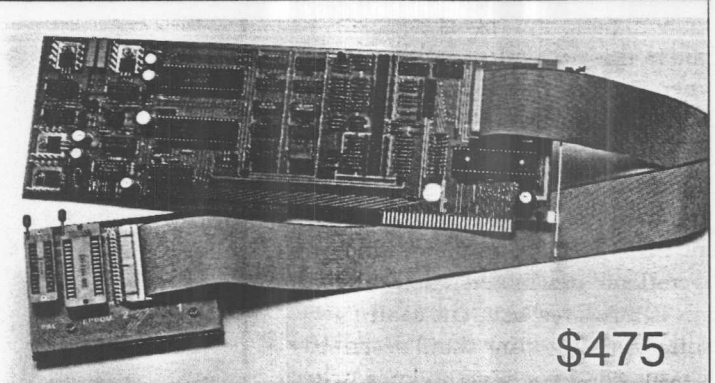
At the bottom of the pile is LCD_OUT. This routine merely passes a couple of arguments from the C code to the assembly level I²C driver. Since I support multiple LCDs and since, as I²C peripherals, they are addressable, this stub routine must determine the appropriate I²C address to pass to the driver routine. The present code supports two LCDs, which is enough to prove a point, but in principle the number could be much higher. You might wonder what I could possibly want with a bunch of LCDs on such a controller. In fact, it turns out I'm already at work on a modular facility surveillance/monitoring system that uses an independent LCD for each zone. As zones are added, a knockout is punched out on the control panel and an indicator display is added.

Since these low-level routines will ultimately be pressed to serve higher-level stream functions such as printf, I had to come up with a way to select the active display device on the fly without too much disruption. This is done by SelectLcd which accepts as its argument a number that denotes which display will become the active LCD and will respond to subsequent function calls. This number is stored and is made globally available for use by the various support routines that need to know. The active display remains live until a different display is selected.

With intelligent peripherals such as LCDs, the initialization sequence is

Universal Device Programmer

PAL
GAL
EPROM
EEPROM
FLASH
MICRO
87C51
PIC
93C46
XC1736
PSD 3xx
5ns PALs

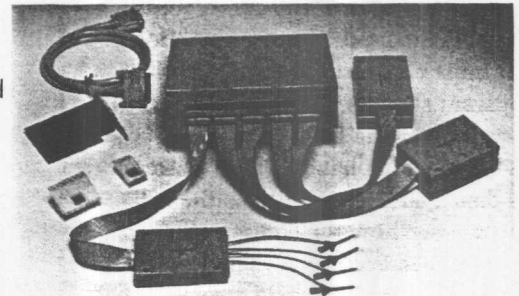


\$475

Free software updates on BBS
Powerful menu driven software

400 MHz Logic Analyzer

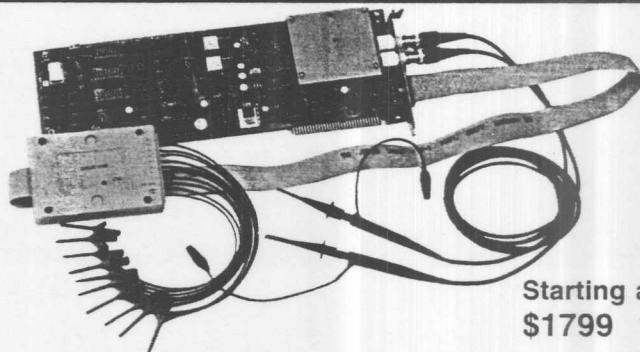
- up to 128 Channels
- up to 400 MHz
- up to 16K Samples/Channel
- Variable Threshold Levels
- 8 External Clocks
- 16 Level Triggering
- Pattern Generator Option



\$799 - LA12100 (100 MHz, 24 Ch)
\$1299 - LA32200 (200 MHz, 32 Ch)
\$1899 - LA32400 (400 MHz, 32 Ch)
\$2750 - LA64400 (400 MHz, 64 Ch)

Price is Complete
Pods and Software
included

200 MSa/s DIGITAL OSCILLOSCOPE



**Starting at
\$1799** With Pods
& Software

- 200 MSa/s sampling Rate
- 2 Analog Channels (2 ch. Digital Oscilloscope)
- 8 Digital Channels (8 ch. Logic Analyzer)
- 125 MHz Single Shot Bandwidth
- up to 128K Samples/Channel

Call (201) 808-8990



Link Instruments

369 Passaic Ave, Suite 100, Fairfield, NJ 07004 fax: 808-8786

undoubtedly the most important step you will perform. This is especially true in this particular implementation where I am using the display in nybble mode. The function `InitLcd` performs the necessary but admittedly arid steps of putting the LCD into a known state, configuring it for four-bit operation, and finally setting up the specific operational parameters. Incidentally, this is, in all respects, the same initialization routine that I described in issue 35. If you want to know what each instruction step actually does to the LCD, refer back to that column.

Since the LCD is operated as a write-only device, the current cursor address is maintained in global RAM. This information is not only needed for keeping track of the cursor for normal cursor positioning operations, but must also be used to preserve the apparent sequential nature of data as it is written to the LCD. If you look at `DataWr`, which handles the transfer of all displayable characters to the LCD, you can see how the cursor must be repositioned at key points where display continuity would otherwise be lost. On entry, the present cursor address is copied to local storage and a switch is used to catch addresses 20, 40, and 60. If the cursor is at any of these addresses, the LCD needs fixing so display data appears where it is expected. Additionally, address 80 is handled in a manner in which the LCD wraps back around to address 0. Finally, before the data character is transferred to the LCD, the current cursor address is incremented.

Now, in preparation for the dual nybble transfers, the data byte is dismembered and positioned. Control bits are appended before the byte is passed to the `LCD_OUT` routine. Since, for displayable data, the destination is the data register, the RS bit is raised to indicate the appropriate target register. Two transfers must be performed for each nybble since the enable (E) signal must be pulsed for each transfer into the LCD. Commands destined for the command register use the `CommandWr` routine, which simply performs the byte unpacking and transfer functions without any of the gyrations `DataWr` has to do. In this case, the command

Listing 2—continued

```
{
    unsigned char b;

    b = Cursor[ActiveLcd];
    switch(b) {
        case 20 :
            CommandWr(64 + 0x80);
            break;
        case 40 :
            CommandWr(20 + 0x80);
            break;
        case 60 :
            CommandWr(84 + 0x80);
            break;
        case 80 :
            CommandWr(0 + 0x80);
            Cursor[ActiveLcd] = 0;
            break;
    }
    Cursor[ActiveLcd]++;
    LCD_OUT((c >> 4) + DEN + DRS);
    LCD_OUT((c >> 4) + DRS);
    LCD_OUT((c & 0xf) + DEN + DRS);
    LCD_OUT((c & 0xf) + DRS);
    return;
}

/* Write to LCD command register */
static void CommandWr(unsigned char c)
{
    LCD_OUT((c >> 4) + DEN);
    LCD_OUT(c >> 4);
    LCD_OUT((c & 0xf) + DEN);
    LCD_OUT(c & 0xf);
    return;
}

/* Initialize LCD panel */
void InitLcd(void)
{
    extern void Delay(char c);
    unsigned char b;
    /* Put LCD in known state */
    Delay(20);
    LCD_OUT(3 + DEN);
    Delay(5);
    LCD_OUT(3);
    Delay(5);
    LCD_OUT(3 + DEN);
    Delay(5);
    LCD_OUT(3);
    Delay(5);
    LCD_OUT(3 + DEN);
    Delay(5);
    LCD_OUT(3);
    Delay(5);
    /* Set 4 bit mode */
    LCD_OUT(2 + DEN);
    LCD_OUT(2);
    Delay(5);
    /* 4 bit, 2 line, 4x7 matrix */
    CommandWr(0x2c);
    /* Display on, cursor off */
    CommandWr(0xc);
    /* Auto increment, shift right */
    CommandWr(0x6);
    /* Clear the LCD and set initial cursor address */
```

(continued)

Listing 2—continued

```

        ClearLcd();
        return;
    }

    /* General delay used by initialization function */
    static void Delay(char c)
    {
        while (c--);
        return;
    }

    /* Select Active LCD */
    void SelectLCD(unsigned char c)
    {
        if (c <= LCDs)
            ActiveLcd = c;
        return;
    }

    /* Assembler linkage: Output a byte over I2C bus */
    static void LCD_OUT(unsigned char c)
    {
        extern void Xmit_I2C_Byte(void);

        if (ActiveLcd == 0) {
            B = c;
            ACC = Lcd0;
            Xmit_I2C_Byte();
        }
        else {
            B = c;
            ACC = Lcd1;
            Xmit_I2C_Byte();
        }
        return;
    }
}

```

register is specified by holding RS low during the transfer sequence.

Cursor positioning consists of writing the cursor address into the appropriate global RAM variable and issuing a cursor positioning command to the LCD's command register. Of course, by now we know that what we see may not be what we get, so `PositionCursor` determines what adjustments must be applied to the cursor address before it can be transferred to the LCD. `ClearLcd` simply issues the clear command to the LCD's command register and zeros the appropriate cursor variable.

Finally, the `putchar` function provides the low-level character output routine that is used by the stream I/O functions. This function is what all the other stuff exists for. All stream routines that output character data do so using the `putchar` function. Newline characters are trapped at

this level, interpreted, and processed in a consistent manner that advances the cursor to the next physical display line. All other characters are simply transferred to the previously described `DataWr`. Note that no checking is performed for valid displayable data at any level of processing described here.

BATTERY CONTROL

If all you're using battery power for is occasional backup, then you can take some liberties in the design of the charging system. On the other hand, if the battery is the primary source of power to your system, it makes sense to devote a little more attention to getting the support circuitry right. This usually results in more power from a given battery and a longer operating life. In the case of the ec.25, the battery manager is centered around the bq2003 fast charge controller IC from Benchmarq Microelectronics.

Battery charging is accomplished by applying constant current with a fall back to a trickle level once a full charge is attained. Negative delta voltage is the method used to determine when fast charging is to be terminated with maximum time used as a failsafe backup. Although the bq2003 fully supports temperature-referenced charge termination, and this is what you'd want to use with a NiMH battery, this capability is not used in my configuration since I'm only supporting NiCds at the present time. Using voltage sensing to terminate fast charge eliminates the added components and interconnects necessary for temperature monitoring and simplifies the battery management circuitry and the battery pack itself.

Figure 2 shows the battery management card. All battery-related activities are controlled by U1, the bq2003 fast charge IC. The bq2003 is configured to control a linear constant-current source and is set up to use negative delta voltage as the charge termination criterium. These two selections are made by respectively pulling SNS to ground and DVEN to V_{cc} . TS and TCO are strapped in a way that inhibits temperature monitoring since it is not used in this configuration. The related TEMP output that is used to indicate temperature status via an LED goes unused also.

The charge initiation input, CCMD, is grounded, which enables automatic battery charging on either application of power or when a battery is connected. Tying TM1 and TM2 to ground selects a negative delta voltage holdoff time of 137 seconds and a fast charge safety time of 360 minutes. The holdoff time is the time after charge initiation that the battery is not monitored for maximum voltage and negative delta voltage. This prevents problems by allowing the battery's charging action to stabilize. The safety time is the maximum time that the battery will be held on fast charge before the charging cycle is terminated. This timeout should never be reached under normal operating conditions, which is precisely what it's all about: safety under abnormal operating conditions.

Although the bq2003 can be used as a switch-mode constant-current source, here I am using it simply as a controller for an external linear current driver. The constant-current source uses a familiar circuit arrangement based on VR2, the popular LM317. The output current of 180 mA is set by R8 and, although not really a fast charge rate, it is adequate to ensure that a typical 600–850-mAH NiCd battery, composed of six AA cells, will attain a full state of charge in a reasonable period of time. This current source is controlled by U1 through switching transistors Q1 and Q2. Once U1 determines that the battery is on its negative voltage slope (negative delta voltage), current from VR2 is cut off. What remains is a small charge-sustaining trickle charge delivered through R4. This trickle resistor has another purpose as well. If the bq2003 determines that the attached battery is at a depleted state

If you look at the battery connector (J1) you will see there are a couple of positive pins: one outgoing and one incoming. When the battery is plugged in, these pins are shorted together since they go to a common connection at the battery's positive terminal. The reason for this is to prevent driving the unloaded output or VR2 (which rises to a relatively high voltage) directly into the main power feed of the system. Recall that last month I mentioned how the MAX639 micro-power switch-mode regulator doesn't do well with anything over 12 V on its input. Using two battery pins, one for input and one for output, is the simplest way of avoiding this type of problem. All you have to do is diode couple all the connections and everybody is happy. Finally, there is an extra unconnected pin on the battery connector for future use.

Having the ability to automatically do a discharge-before-charge on a NiCd battery can be particularly useful in preventing and correcting the voltage depression effect popularly known as "memory." An on-demand discharge cycle is started by pressing SW1, a momentary push-button switch that feeds into DCMD. When U1 detects a positive edge on DCMD,

Next month: an I²C infestation.

John Dybowski is an engineer involved in the design and manufacture of hardware and software for industrial data collection and communications equipment. He may be reached at john.dybowski@circellar.com.

For elements of this project, contact:

Mid-Tech Computing Devices
P.O. Box 218
Stafford, CT 06075-0218
(203) 684-2442

419 Very Useful
420 Moderately Useful
421 Not Useful

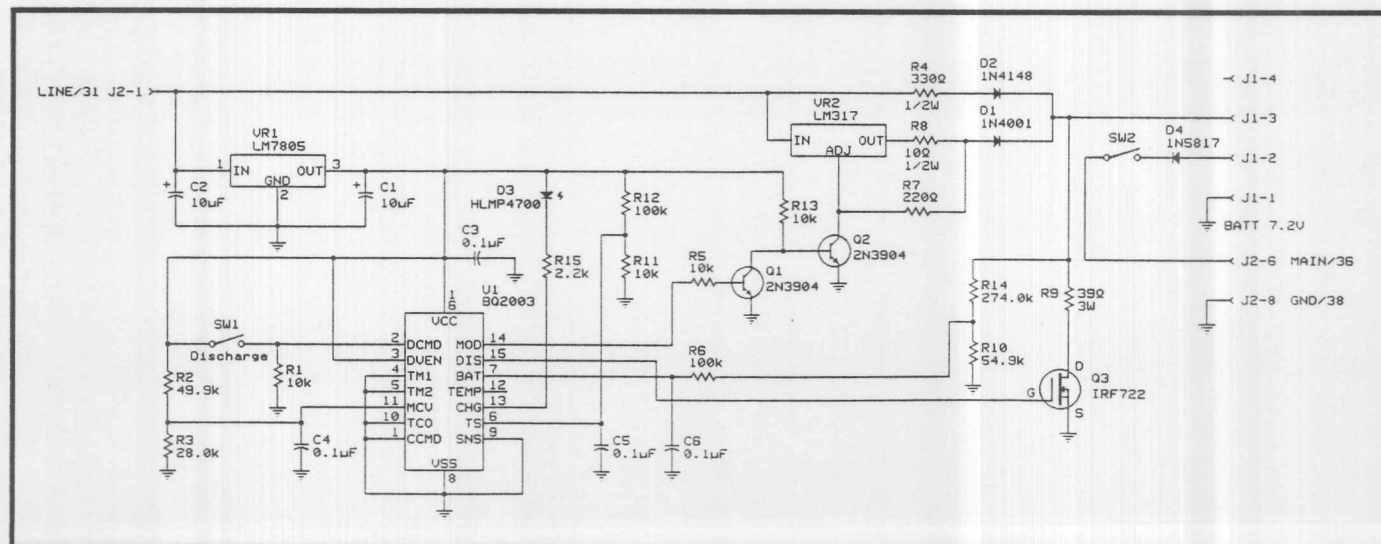


Figure 2—The battery management card specifically handles NiCds, but could handle other kinds of batteries with minor changes.